Homework 3 Selected Solutions
APPM/MATH 4650 Fall ’20 Numerical Analysis

Due date: Friday, September 18, before 5 PM, via Gradescope. **Instructor**: Prof. Becker
Theme: Newton’s method and rootfinding

solutions version 9/16/2020

Problem 1: Which of the following iterations will converge to the indicated fixed point p (provided x_0 is sufficiently close to p)? If it does converge, give the order of convergence; for linear convergence, give the rate of linear convergence.

a) $x_{n+1} = -16 + 6x_n + \frac{12}{x_n}$, $p = 2$

Solution:

This is the fixed point iteration for the function $g(x) = -16 + 6x + 12/x$. We can confirm that p is a fixed point, since $g(p = 2) = -16 + 12 + 6 = 2$.

We need $|g'(p)| < 1$ for the iteration to converge. The derivative is $g'(x) = 6 - 12/x^2$. So $|g'(p)| = |6 - 12/p^2| = 6 - 3 > 1$. Therefore the iteration will $\boxed{\text{not converge}}$ (well, as a lawyer might say, we have no evidence that it will converge, though to be precise we did not *prove* that it diverges; for example, not only starting at $x_0 = 2$ but also at $x_0 = 1$ will lead to $x_1 = p = 2$ and hence the sequence converges for these very “lucky” starting values).

b) $x_{n+1} = \frac{2}{3}x_n + \frac{1}{x_n^2}$, $p = 3^{1/3}$

Solution:

This is the fixed point iteration for the function $g(x) = 2/3x+1/x^2$. We need $|g'(p)| < 1$ for the iteration to converge. The derivative is $g'(x) = 2/3-2/x^3$. So $|g'(p)| = |2/3-2/3| = 0$. Therefore the iteration $\boxed{\text{will converge}}$.

In order to determine the order of convergence, we can use Theorem 2.9, which shows that if $|g''|$ is bounded near p, then the convergence is quadratic. This is easy to confirm, as g'' is continuous away from $x = 0$, so since $p \neq 0$, g'' can be bounded near p, hence we have $\boxed{\text{quadratic convergence}}$.

Another way to see this (i.e., looking at the proof of Theorem 2.9), we can look at Taylor’s theorem with expanding about p, and see that there exist an ξ in the interval between p and x such that

$$\begin{aligned} g(x) &= g(p) + g'(p)(x-p) + \frac{g''(\xi)}{2!}(x-p)^2 \\ &= p + 0 + \frac{g''(\xi)}{2!}(x-p)^2 \end{aligned}$$

so if $|g''(x)| \leq M$ for x sufficiently close to p, then $|g(x) - p| \leq M/2|x-p|^2$. Letting $x = x_n$, then $g(x) = x_{n+1}$, so

$$|x_{n+1} - p| \leq \left|\frac{M}{2}\right| |x_n - p|^2$$

thus the converge is quadratic. [This full derivation was not needed by students]

c) $x_{n+1} = \frac{12}{1+x_n}$, $p = 3$

Solution:

This is the fixed point iteration for the function $g(x) = \frac{12}{1+x}$. Thus $g'(x) = -\frac{12}{(1+x)^2}$. Plugging in p, we find $|g'(p)| = 12/16 < 1$. Thus the iteration $\boxed{\text{converges linearly}}$ according to Theorem 2.8.

To determine the constant, we use Taylor's theorem taking the expansion about p which states there exist an ξ_n between p and x_n such that

$$g(x_n) = g(p) + g'(\xi_n)(p - x_n).$$

We know $g(p) = p$, $g(x_n) = x_{n+1}$ and $\lim_{n\to\infty} x_n = p$.

Thus $\lim_{n\to\infty} \frac{|x_{n+1}-p|}{|x_n-p|} = \lim_{n\to\infty} |g'(\xi_n)| = 12/16$ since $x_n \to p$ and $0 \le \xi_n \le x_n$ implies $\xi_n \to p$ by the squeeze theorem. So the rate is $\boxed{12/16}$ i.e., $\boxed{3/4}$. Note: students were not expected to give a formal Taylor theorem + squeeze theorem proof; it was sufficient to note $g'(p) = 12/16$ and $g \in C^2$ in an interval around p.

In general, if g is $C^2(I)$ for some open interval I with $p \in I$, and f has bounded second derivative, then if $|g'(p)| < 1$, it will converge (if we start close enough) and if $|g'(p)| \ne 0$ then it is linear convergence at rate $|g'(p)|$. If $g'(p) = 0$ then (assuming g'' is bounded nearby) we have quadratic convergence. These are the arguments we used in the proof of Theorems 2.6, 2.8 and 2.9 (these theorems all have a similar feel, and the *core* idea is still the contraction mapping theorem).

Another way to evaluate $\lim_{n\to\infty} \frac{|x_{n+1}-p|}{|x_n-p|}$ is to write (noting that we already proved linear convergence, so we know $x_n \to p = 3$)

$$\begin{aligned}
\lim_{n\to\infty} \frac{x_{n+1}-3}{x_n-3} &= \lim_{n\to\infty} \frac{g(x_n)-3}{x_n-3} \\
&= \lim_{x\to 3} \frac{\frac{12}{1+x}-3}{x-3} \left(= \frac{0}{0}\right) \\
&\overset{\text{L'H}}{=} \lim_{x\to 3} \frac{\frac{-12}{(1+x)^2}}{1} \\
&= \frac{-12}{16} = \frac{-3}{4}
\end{aligned}$$

so when we take absolute values, we get $\frac{3}{4}$.

Problem 2: Programming Newton's Method Program Newton's method in a language of your choice, and make sure you can save the history of all the variables x_n and their values $f(x_n)$. You may include your code as an appendix if you want, but grading of this will be based on the following specific items:

a) We will test that our code is working by using a function with a known root. Using $f(x) = (x-3)\cdot(x-2)$, first program the derivative f'. This is an easy derivative, but for more complex functions, it's easy to either make a math mistake or a typing error when programming the derivative. Hence, we want to check that your derivative is implemented correctly. **Plot** your derivative on the interval $[1, 4]$, and then plot a finite difference approximation of the derivative (using only your function f, *not* f') on the same plot. Ensure that these plots are close enough. For finite difference approximations, using `diff` (Matlab) or `numpy.diff` (Python) are useful.

Solution:

Python code is Ch2_NewtonsMethod.ipynb on the Master branch of our class github site.

Basic Matlab code is below, and a fancier version is given at the end of this PDF.

```matlab
function [x,iter,x_hist,fx_hist] = newton(f,fprime,x)
% [x,iter,x_hist,fx_hist] = newton(f,fprime,x0)
%   runs Newton's method on the function f starting at x0
MaxIter     = 50;
for iter= 1:MaxIter
    fx      = f(x);          % Evaluate function
    fprimex = fprime(x);     % Evaluate derivative
    fprintf('Iter %3d, x_n is %19.16f, f(x_n) is %+5.2e, f''(x_n) is %+5.2e\
        n',iter,x,fx,fprimex);
    if iter < MaxIter
        x   = x - fx/fprimex;   % Take the Newton step
    end
end
```

Actual code for the finite different approximation (in Matlab) is below, and the corresponding plot is in Fig. 1.

```matlab
f       = @(x) (x-3).*(x-2); % x^2 - 5x + 6
fprime  =  @(x) 2*x - 5;
xGrid   = linspace(1,4,200);
h       = xGrid(2)-xGrid(1);
plot( xGrid, fprime(xGrid),'linewidth',1.5,'DisplayName','f''(x)');
hold all
plot( xGrid(2:end), diff(f(xGrid))/h,'--','linewidth',3,'DisplayName','
    Finite diff approximation of f''(x)');
set(gca,'fontsize',24);
legend
export_fig HW3_Problem2a -pdf -transparent
```

Note: for both Python and Matlab, the output of `diff` is always 1 smaller than the input, e.g., if $x = \{1,2,3,4\}$ (which has length 4) then the output of `diff(x)` is $\{1,1,1\}$ which has length 3. So for plotting, you need to adjust things to make the length work out. In the code above, we did this by shortening `xGrid` to exclude its first entries; you could just as well exclude the last entry, or do both and average the answer.

b) Still using $f(x) = (x-3)\cdot(x-2)$, give a **plot** for the error $|x_n - 3|$ for $x_0 = 10$ using your Newton's method code. **Discuss**: does this error decay to zero, and does it decay at the rate you expect?

Solution:

We see fast convergence when the y-axis is logarithmic, indicating this is faster than linear convergence (and in fact $\boxed{\text{quadratic}}$). This is what we expect from Newton's method (when it converges) since $x = 3$ is a $\boxed{\text{simple root}}$. The plot is in Fig. 1. Simplified Matlab code is:

```matlab
f       = @(x) (x-3)*(x-2); % x^2 - 5x + 6
fprime  =  @(x) 2*x - 5;
[x,iter,x_hist,fx_hist] = newton(f,fprime,x0)
semilogy( abs(x_hist - 3)
```

c) Now use $f(x) = (x-3)^2$, and give a **plot** for the error $|x_n - 3|$ for $x_0 = 10$ using your Newton's method code. **Discuss**: does this error decay to zero, and does it decay at the rate you expect?

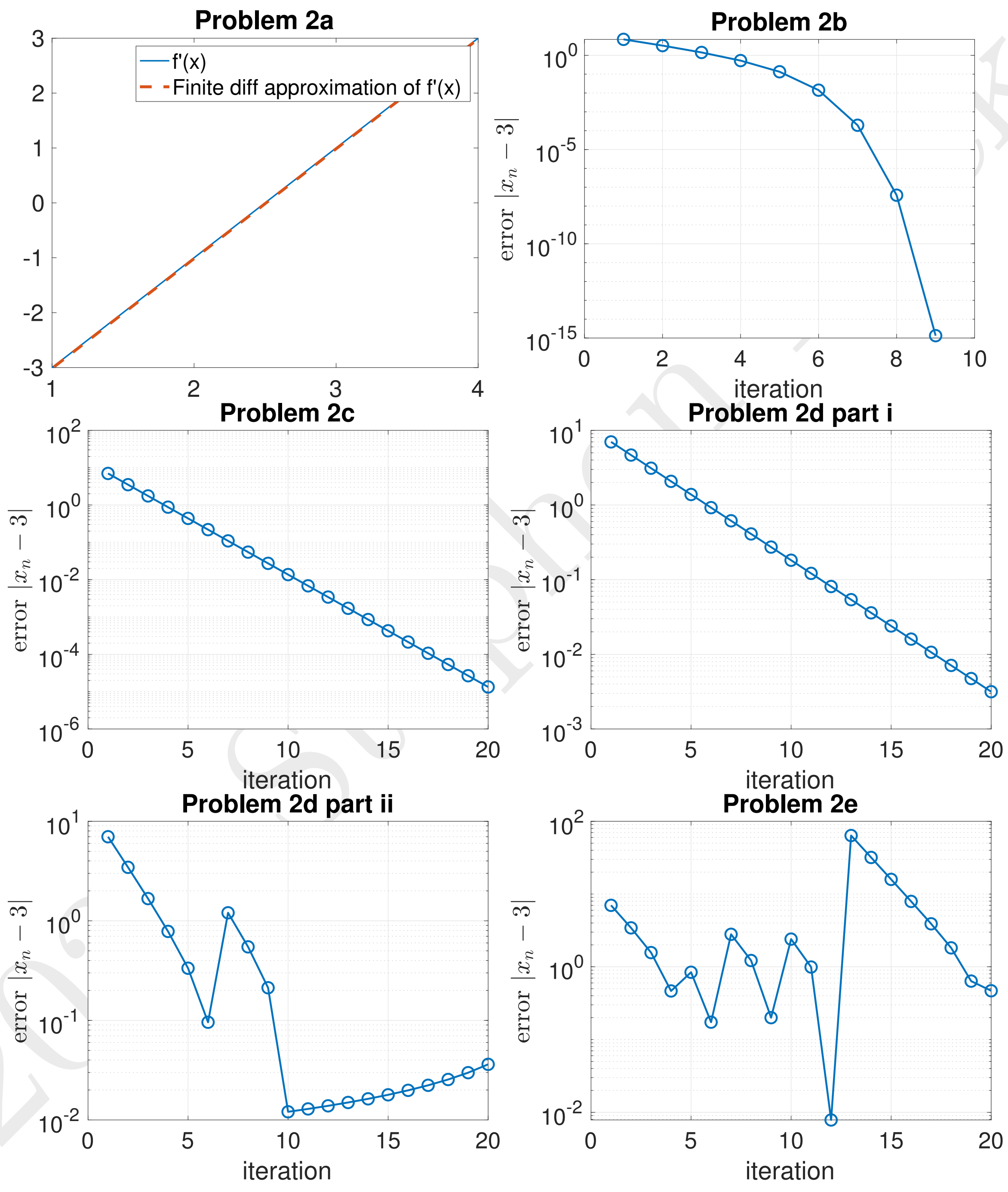

Figure 1: Plots for Problem 2

Solution:

We see a straight line when the y-axis is logarithmic, indicating this is $\boxed{\text{linear convergence}}$. This is what we expect from Newton's method (when it converges) since $x = 3$ is a $\boxed{\text{double root}}$. The plot is in Fig. 1. Simplified Matlab code is:

```
f      = @(x) (x-3)^2;
fprime = @(x) 2*(x-3);
[x,iter,x_hist,fx_hist] = newton(f,fprime,x0)
semilogy( abs(x_hist - 3)
```

d) Misspecify the derivative for $f(x) = (x-3)^2$. The true derivative is $2(x-3)$, but misspecify it as (i) $3(x-3)$ and (ii) $2(x-3.1)$. For both cases, **describe** what happens to the error.

Solution:

The plot is in Fig. 1. For the first case, Newton's method still seems to converge, just slightly more slowly. This might be surprising! In fact, if we misspecified $g'(x) = 1$, then Newton's method is the algorithm called gradient descent, which converges under many conditions (though you need a *stepsize* to guarantee convergence). Gradient descent almost never converges quadratically.

We can be more specific (not necessary for student answers): with this misspecified derivative, we're running a fixed point iteration $x_{n+1} = g(x_n)$ for $g(x) = x - \frac{(x-3)^2}{3(x-3)}$, so then $g'(3) = 1 - \frac{1}{3} = \frac{2}{3}$, and hence we can see that we still have a contraction, and will converge at linear rate with constant $\frac{2}{3}$. With the true derivative, we would have converged at a linear rate with constant $\frac{1}{2}$ (not quadratically, since it's not a simple root).

For the second case, the method does not appear to converge. This is not so surprising since afterall we did not give it the correct derivative. We can be more specific (not necessary for student answers): with this misspecified derivative, we're running a fixed point iteration $x_{n+1} = g(x_n)$ for $g(x) = x - \frac{(x-3)^2}{(x-3.1)}$ so $g'(x) = 1 - \frac{1}{2}\frac{2(x-3)\cdot(x-3.1)-(x-3)^2}{(x-3.1)^2}$. So $g'(3) = 1$ and this is not a contraction, so we don't have any reason to believe it will converge.

e) Now use $f(x) = (x-3)^2 + 1$ which does not have a real root. **Discuss**: what happens to Newton's method?

Solution:

The plot is in Fig. 1. Newton's method does not converge. When we detect this oscillatory behavior, it's hard to know if it is due to numerical error or because there is no root; this kind of observed behavior can also mean that there is a root but we just didn't start close enough to it.

Problem 3: Modified Newton's Method Let $f(x) = (x - 1/3)^5$.

a) Run Newton's method, starting at $x_0 = 0.4$, and include either a list (up to 20 iterations) or a plot of the errors. Does the error decay as expected?

Solution:

We see a straight line when the y-axis is logarithmic, indicating this is $\boxed{\text{linear convergence}}$. This is what we expect from Newton's method (when it converges) since $x = 3$ is a $\boxed{\text{not a simple root}}$. The plot is in Fig. 2. Simplified Matlab code is:

```
p = 1/3;
f      = @(x) (x-p).^5;
fprime = @(x) 5*(x-p).^4;
% ... or, you can define f and fprime as:
coeff  = poly( ones(1,k)*p );
```

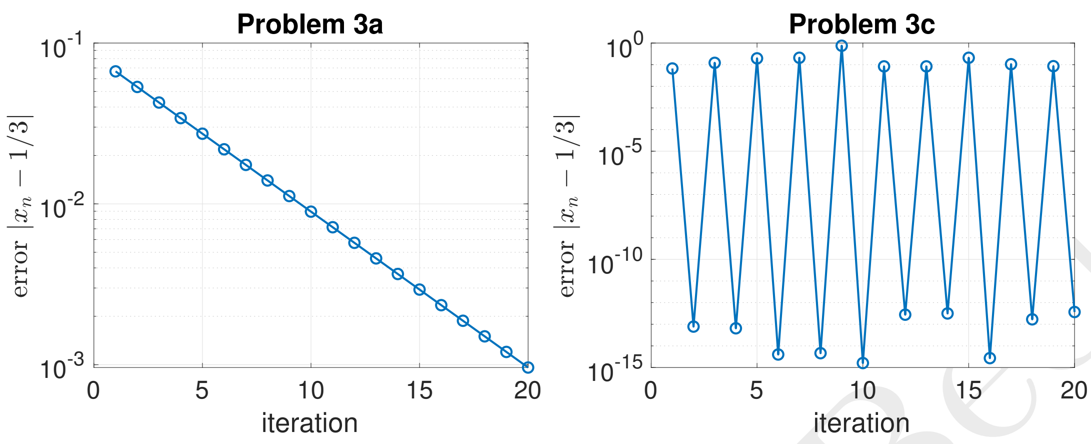

Figure 2: TBD

```
f       = @(x) polyval( coeff, x );
fprime  = @(x) polyval( polyder(coeff), x );

x0 = .4;
[x,iter,x_hist,fx_hist] = newton(f,fprime,x0,'Print',true,'TolX',1e−12,'
    TolFun',1e−20);
```

b) Now program Modified Newton's method, i.e., run Newton's method on $\mu(x) = f(x)/f'(x)$ since $\mu(p) = f(p) = 0$ but $\mu'(p) \neq 0$, still starting at $x_0 = 0.4$. Simplify the expression for $\mu(x)$ by hand. Include a list of the error. Is this expected?

Solution:

Simplifying μ gives $\mu = 1/5(x - 1/3)$. We won't include a plot because there's not much to show: deflation works so well that we converge after one Newton step. In fact, this is expected, since μ is a linear function. Newton's method works by using a linear approximation of the function (the Taylor series, i.e., tangent line)... but in fact, the linear approximation of a linear function is not an approximation, it is exact! So that's why Newton's method converges in one step. (Well, in our code, it seems to be 2 steps, but that's just because Iter 1 was displaying what the input was).

Matlab code:

```
mu = @(x) 1/5*(x−p);
muprime = @(x) 1/5;
[x,iter,x_hist,fx_hist] = newton(mu,muprime,x0,'Print',true,'TolX',1e−12,'
    TolFun',1e−12);
```

and the output:

```
Iter   1, x_n is  0.4000000000000000, f(x_n) is +1.33e-02, f'(x_n) is +2.00e-01
Iter   2, x_n is  0.3333333333333333, f(x_n) is +0.00e+00, f'(x_n) is +2.00e-01
Stopping (reached tolerance or NaN)
```

Unfortunately, this is not a realistic problem, since we wouldn't ever have f in factored form (if we did, we would be able to just read off the root).

c) Now program Modified Newton's method again, but implement $\mu(x)$ explicitly as the ratio of $f(x)$ over $f'(x)$ and use `polyval` (Matlab) or `numpy.polyval` (Python) to evaluate $f(x)$, $f'(x)$ and $f''(x)$. To find the coefficients of f, you can use `poly` (Matlab) or `numpy.poly` (Python) and specify that the root is at 1/3, and to find the coefficients of the derivative, you can use `polyder` (Matlab) or `numpy.polyder` (Python). Do the quotient rule to determine $\mu'(x)$ in terms of f, f' and f'' (do *not* use knowledge of what f is in order to simplify this). Include a list of the first 20 error terms. Is this expected?

Solution:

A Matlab implementation is (note the use of `.*` and `./` and `.^2`, which tell Matlab that these operations should be vectorized, so that later we can apply a whole list of inputs rather than having to make a `for` loop; in Python with numpy, you don't have to do this):

```
fdoubleprime = @(x) polyval( polyder(polyder(coeff)), x );
mu = @(x) f(x)./fprime(x);
muprime = @(x) ( fprime(x).^2 − f(x).*fdoubleprime(x) )./(fprime(x).^2 );
[x,iter,x_hist,fx_hist] = newton(mu,muprime,x0,'Print',true,'TolX',1e−12,'
    TolFun',1e−12);
```

and a list of output is below (and a plot in Fig. 2):

```
Iter   1, x_n is  0.4000000000000000, f(x_n) is +1.33e-02, f'(x_n) is +2.00e-01
Iter   2, x_n is  0.3333333333332553, f(x_n) is +8.75e-02, f'(x_n) is +7.20e-01
Iter   3, x_n is  0.2118055555554775, f(x_n) is -2.43e-02, f'(x_n) is +2.00e-01
Iter   4, x_n is  0.3333333333332682, f(x_n) is +1.41e-01, f'(x_n) is +7.19e-01
Iter   5, x_n is  0.1376811594202247, f(x_n) is -3.91e-02, f'(x_n) is +2.00e-01
Iter   6, x_n is  0.3333333333333293, f(x_n) is +2.08e-01, f'(x_n) is +1.00e+00
Iter   7, x_n is  0.1249999999999960, f(x_n) is -4.17e-02, f'(x_n) is +2.00e-01
Iter   8, x_n is  0.3333333333333287, f(x_n) is +2.50e-01, f'(x_n) is -3.33e-01
Iter   9, x_n is  1.0833333333333286, f(x_n) is +1.50e-01, f'(x_n) is +2.00e-01
Iter  10, x_n is  0.3333333333333317, f(x_n) is +8.33e-02, f'(x_n) is +1.00e+00
Iter  11, x_n is  0.2499999999999984, f(x_n) is -1.67e-02, f'(x_n) is +2.00e-01
Iter  12, x_n is  0.3333333333330559, f(x_n) is +8.33e-02, f'(x_n) is +1.00e+00
Iter  13, x_n is  0.2499999999997226, f(x_n) is -1.67e-02, f'(x_n) is +2.00e-01
Iter  14, x_n is  0.3333333333330176, f(x_n) is +2.08e-01, f'(x_n) is +1.00e+00
Iter  15, x_n is  0.1249999999996842, f(x_n) is -4.17e-02, f'(x_n) is +2.00e-01
Iter  16, x_n is  0.3333333333333306, f(x_n) is +1.46e-01, f'(x_n) is +1.39e+00
Iter  17, x_n is  0.2283333333333306, f(x_n) is -2.10e-02, f'(x_n) is +2.00e-01
Iter  18, x_n is  0.3333333333331644, f(x_n) is +7.50e-02, f'(x_n) is +8.80e-01
Iter  19, x_n is  0.2481060606058917, f(x_n) is -1.70e-02, f'(x_n) is +2.00e-01
Iter  20, x_n is  0.3333333333329646, f(x_n) is +1.75e-01, f'(x_n) is +4.40e-01
```

What's going on?!? It looks like we almost converged very quickly, but then we moved away. At the next step we get very close again, only to move away. The issue is that μ is unstable near the root, so floating point error messes up the algorithm. In particular, if we define μ the *nice* way (from the previous problem) and evaluate it at 1/3, then we get `mu(1/3)=0` as we are guaranteed (with exact arithmetic). But now if we define it this not-so-nice way using coefficients (less accurate due to cancellation, but more realistic) we get `mu(1/3)=.2083` which is very incorrect. We ought to get 0 using exact arithmetic, but we do not.

The moral of the story is that Modified Newton's method (as well as other tricks like deflation) may not work so well on some problems, due to instability. The problems that need these tricks are inherently ill-conditioned, so there is not always a clever algorithm that can find them with full accuracy.

Another way to see the issue is that we technically define μ to be the following function (in general, where p is the root):

$$\mu(x) = \begin{cases} \frac{f(x)}{f'(x)} & x \neq p \\ 0 & x = p \end{cases}$$

which is continuous (as can be shown via L'Hôptal's rule). But on a computer, we get issues when $x \approx p$.

d) Repeat (b) and (c) for the polynomial $f(x) = (x - 1/2)^5$. Comment on any differences, and speculate about why they may arise.

Solution:

For the simplified μ, things work nicely, just as they did before:

```
Iter   1, x_n is  0.4000000000000000, f(x_n) is -2.00e-02, f'(x_n) is +2.00e-01
Iter   2, x_n is  0.5000000000000000, f(x_n) is +0.00e+00, f'(x_n) is +2.00e-01
```

However, when we try the not-so-nice version of μ that has more roundoff error, it actually works OK

```
Iter   1, x_n is  0.4000000000000000, f(x_n) is -2.00e-02, f'(x_n) is +2.00e-01
Iter   2, x_n is  0.4999999999999057, f(x_n) is  +Inf, f'(x_n) is   NaN
Stopping (reached tolerance or NaN)
```

Why does it suddenly work? We can check that even if μ is defined in this not-so-nice way, `mu(.5)=NaN` because we have a $0/0$. Even though there is roundoff error in general, because the coefficients are integers and can be represented without any error in base 2, and because $p = 1/2$ can also be represented exactly in base 2, this means that at the exact value of $x = 1/2$ (or anything within machine epsilon of this), the formula is accurate. Because Newton converges so quickly, it only took us one step to reach this machine epsilon accuracy of $x = 1/2$.

In fact, had we started at $x_0 = 1/4$ (which is also exactly representable in binary), we might have have found in Iteration 2 that $x_n = 0.5$ exactly! If you ever want to check if a number has an exact representation in binary (floating point), then this website is useful; just enter your decimal number, and see if the output has the "inexact" flag checked.

Note the following behavior which indicates just how sensitive this μ implementation is near the root:

```
>> mu(.5)
NaN
>> mu(.5+eps)  % eps is machine epsilon, 2.2204e-16
Inf
>> mu(.5+2*eps)
0
>> mu(.5+3*eps)
0.25
```

If we implement μ the nice way, then we get the kind of numbers that we ought to get:

```
>> mu(.5)
0
>> mu(.5+eps)
4.4409e-17
>> mu(.5+2*eps)
8.8818e-17
>> mu(.5+3*eps)
1.3323e-16
```

Problem 4: Word problem Salma is a dog lover and wants to own dogs. Her happiness level for having d dogs is given by the function

$$F(d) = \underbrace{d^2}_{\text{joy of having dogs}} - \underbrace{.5\left(e^d - 1\right)}_{\substack{\text{unpleasantness of}\\ \text{picking up dog waste}}} .$$

How many dogs should she own in order to maximize her happiness? (Assume that fractional amounts of dogs are allowed, e.g., either counting small dogs as part of a whole dog, or by

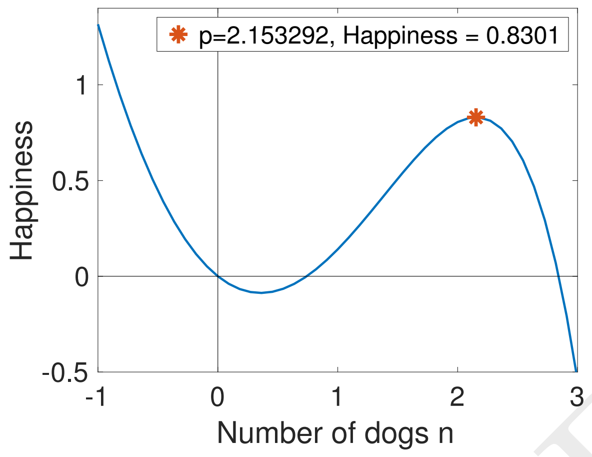

Figure 3: Graph for Problem 4

sharing dog ownership with a friend). Specifically, show how to convert this into a root-finding problem, solve it with your own root-finding implementation (include some output from every iteration), and report the final answer.

Solution:

This is a minimization problem. Scalar minimization problems can be solved by checking the end point (here, this is $d = 0$ since we need non-negative dogs, as well as $d \to \infty$) and any points where the function is discontinuous or non-differentiable (for this problem, there are no such points), and any critical points where $F' = 0$. For $d = 0$, we observe $F(0) = 0$. Now we check critical points, so we solve the rootfinding problem $F'(d) = 2d - .5e^d = 0$. We can solve this via Newton's method, using $f(d) = F'(d)$, and calculating $f'(d) = F''(d) = 2 - .5e^d$. Looking at a plot like Fig. 3, we guess that $d = 2$ is a reasonable starting point, and run Newton's method. Our output is

```
Iter   1, x_n is  2.0000000000000000, f(x_n) is +3.05e-01, f'(x_n) is -1.69e+00
Iter   2, x_n is  2.1802696335602474, f(x_n) is -6.38e-02, f'(x_n) is -2.42e+00
Iter   3, x_n is  2.1539505125873188, f(x_n) is -1.52e-03, f'(x_n) is -2.31e+00
Iter   4, x_n is  2.1532927681634013, f(x_n) is -9.32e-07, f'(x_n) is -2.31e+00
Iter   5, x_n is  2.1532923641105017, f(x_n) is -3.51e-13, f'(x_n) is -2.31e+00
```

and conclude that $\boxed{d^\star = 2.153292}$ is the appropriate number of dogs to own. To be careful, we do need to check our boundaries, meaning that we check $F(d^\star) > F(0)$ and $F(d^\star) > \lim_{d\to\infty} F(d)$. Since $F(d^\star) \approx 0.8301$ and $F(0) = 0$ and $\lim_{d\to\infty} F(d) = -\infty$, this checks out.

Here is a more complete version of the `newton.m` code for Problem 2:

```
function [x,iter,x_hist,fx_hist] = newton(f,fprime,x,varargin)
% [x,iter,x_hist,fx_hist] = newton(f,fprime,x0)
%   runs Newton's method on the function f starting at x0
prs = inputParser; % optional fancy stuff
addParameter(prs,'Print',false);  % display output every iteration
addParameter(prs,'MaxIter',20);   % same name as in optimset
addParameter(prs,'TolFun',1e-10); % same name as in optimset
addParameter(prs,'TolX',1e-10);   % same name as in optimset
parse(prs,varargin{:});
MaxIter     = prs.Results.MaxIter;
printInfo   = prs.Results.Print;
TolFun      = prs.Results.TolFun;
TolX        = prs.Results.TolX;
```

```



[x_hist,fx_hist] = deal( zeros(1,MaxIter) );
for iter= 1:MaxIter

    fx      = f(x);         % Evaluate function
    fprimex = fprime(x);    % Evaluate derivative

    % Book-keeping, printint out stuff, etc.
    if printInfo
        fprintf('Iter %3d, x_n is %19.16f, f(x_n) is %+5.2e, f''(x_n) is %+5.2e\n',iter,x,fx
            ,fprimex);
    end
    x_hist(iter)  = x; fx_hist(iter) = fx; % save history
    if abs(fx) < TolFun || ...
            (iter>1 && abs(x-x_hist(iter-1))/max(abs(x),1e-16) < TolX ) ...
            || isnan(fx) || isnan(fprimex)
        x_hist = x_hist(1:iter); fx_hist = fx_hist(1:iter);
        if printInfo, fprintf('Stopping (reached tolerance or NaN)\n'); end
        break
    end
    if iter < MaxIter
        % Take the Newton step
        x   = x - fx/fprimex;
    end
end
```